Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 178 (2007) 161–169
www.elsevier.com/locate/entcs

A General Framework for Overlay Visualization

Tihomir Piskuliyski[1]

Ramapo College of New Jersey
505, Ramapo Valley Road
Mahwah, NJ, USA

Amruth Kumar[2]

Ramapo College of New Jersey
505, Ramapo Valley Road
Mahwah, NY, USA

Abstract

If visualization is more effective when accompanied by narration, why not superimpose visualization on narration? This might result in better transfer of learning. We will present a general framework for such superimposed visualization, called overlay visualization. The objectives for the design of our framework are 1) to separate the application from the visualization; and 2) to separate the specification from the rendering. We will describe a few applications of overlay visualization for programming and provide examples from our implementation of overlay visualization for software tutors called problets. The advantages of overlay visualization include: less cognitive load on the learner, and automatic support for both path and state visualization.

Keywords: Overlay visualization, superimposed visualization, program visualization

1 Introduction

Researchers have found that on problem-solving transfer tasks [10] animation with narration outperforms animation only, narration only, or narration before animation. Similarly, on recall tasks, narration with visual presentation outperforms narration before visual presentation [1]. These results support a dual-coding hypothesis [13] that suggests two types of connections among stimuli and representations: representational connections between stimuli and the corresponding representations

[1] Email: tpiskuli@ramapo.edu
[2] Email: amruth@ramapo.edu

doi:10.1016/j.entcs.2007.01.046

(verbal and visual), and referential connections between verbal and visual representations.

What if visual presentation is superimposed on narration? In programming problems, what if visual presentation is superimposed on the program code? We conjecture that this will promote better referential connections between visual and verbal representations and result in better transfer from visual representation to the concepts being learned. To support such visualization, we have developed a framework of what we will henceforth refer to as overlay visualization.

Overlay Visualization is the superimposition of graphics on the material to be visualized. In the context of programming, it is superimposing graphics on code. We will first describe a general framework for overlay visualization in section 2. Next, in section 3, we will discuss some applications of overlay visualization for program visualization. We will illustrate with examples from our implementation of overlay visualization for programming tutors called problets (`www.problets.org`) [6]. In section 4, we will present the current implementation. Finally in section 5, we will discuss the advantages of overlay visualization and compare it with prior work.

2 A General Framework for Overlay Visualization

Objectives: We had two objectives for our overlay visualization framework: 1) to separate the application from the visualization; and 2) to separate the specification from the rendering.

While separating the application from the visualization, we wanted to ensure two objectives: 1) maximize the flexibility of visualization; and 2) minimize the overhead of specifying such visualization. Separating the specification from the rendering and using a declarative representation for the specification increases the flexibility of the framework. The specification can be coded by hand, generated automatically by a program, or specified by the learner using mouse gestures. In all these cases, the visualizer that translates the specification into visualization would be the same.

We identified two layers of separation between the application and the visualization: a layer of visual primitives and a layer of graphical primitives.

Graphical Primitives: The graphical primitives that we have implemented so far are: 1) Box - draws/animates a box, 2) Arrow - draws/animates an arrow, 3) Ellipse - draws/animates an ellipse, 4) Text - draws/animates a string of text.

Visual Primitives: The visual primitives that we have identified and implemented so far are: 1) Point at certain words or lines of code using arrows, 2) Highlight lines of feedback and/or code using boxes, 3) Circle lines of feedback and/or code using ellipses, 4) Connect two circled segments of code with an arrow, and 5) Animate a given segment of text.

Visual Specification: A visual specification consists of the type of the visual

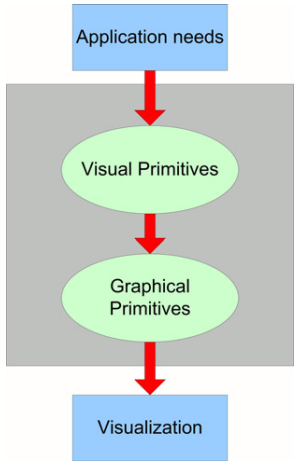

Fig. 1. The architecture of the overlay visualization system

primitive, the line in the code for which the visual primitive needs to be created, whether the visual primitive should be drawn or animated, the order in which the visual primitives must be drawn/animated, and the color in which the visual primitive must be rendered. Since the visual specification is declarative in nature, it can be generated in several ways, as mentioned earlier:

- **Automatically generated by the application:** The application that uses the overlay visualization can automatically determine the lines of code that must be visualized, the visual primitives with which they should be visualized, and the order in which they should be visualized, and generate the visual specifications accordingly.
- **Specified by the learner:** The learner can use mouse gestures to specify the visual primitives that should be used, the lines of code for which the primitives should be used, and the order in which they should be rendered.

Visualizer: The visualizer (shown as a dark box in Figure 1) gets a declarative list of visual specifications as input, e.g., a list of specifications to illustrate the execution of a `for` loop, `while` loop, or `switch` statement. The Visualizer creates the visual primitives corresponding to the visual specifications. The visual primitives in turn create the graphical primitives needed to render them. During rendering, the Visualizer sequences the rendering of the visual primitives, which in turn delegate the rendering to their corresponding graphical primitives.

3 Applications of Overlay Visualization

Overlay visualization can be used for various purposes in program visualization: to visualize the control flow in a program, visualize the data flow in a program, cluster the code by functionality, set off missing stages in the lifetime of a variable, etc. In each case, the visualization serves to focus the attention of the learner on the segments of code that are of immediate interest. We will present examples of

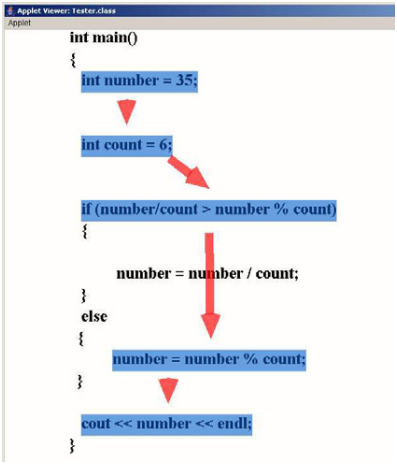

Fig. 2. An Example of Control Flow Visualization

control flow visualization, data flow visualization and error highlighting from our implementation of overlay visualization for programming tutors called problets.

Control flow Visualization clarifies the order in which the lines of code in a program are executed. Control flow visualization is especially helpful when the program involves selection statements, repetition statements and function calls. An example of control flow visualization generated by the overlay visualizer is shown in Figure 2. The specification provided to the visualizer was:

(i) CONNECT line 2 to line 5
(ii) CONNECT line 5 to line 8
(iii) CONNECT line 8 to line 15
(iv) CONNECT line 15 to line 18

We plan to have problets automatically generate this specification as they execute the program, by keeping tracking of the line numbers of the lines of code that are executed.

Setting off an error: When a program object does not go through the correct sequence of state transitions, it may end up in an error state, e.g., a variable is not assigned before it is referenced, a pointer is not allocated before it is de-referenced, or a loop is not updated in its body. Overlay visualization can be used to highlight or set off the missing state transition and clarify the origin of the error.

Data flow visualization clarifies the sequence of transformations applied to one or more variables in a program. Data flow visualization is especially helpful when operations are applied to a variable in non-contiguous sections of a program. It complements data space visualization provided in systems such as Jeliot 3 [2] and problets [7] - whereas data space visualization provides a snapshot of the latest values of all the variables in a program, data flow visualization displays the sequence

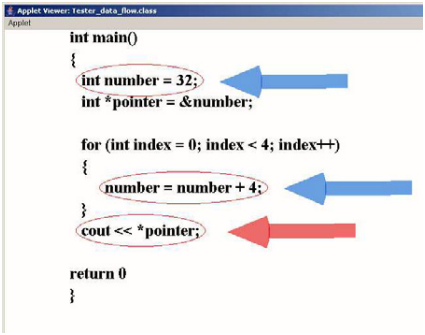

Fig. 3. An Example of Data-Flow Visualization

of operations performed on one or more variables that can explain their latest value. An example of data flow visualization generated by the overlay visualizer is shown in Figure 3. The specification provided to the visualizer was:

(i) CIRCLE line 2 in red
(ii) POINT to line 2 in blue
(iii) CIRCLE line 7 in red
(iv) POINT to line 7 in blue
(v) CIRCLE line 9 in red
(vi) POINT to line 9 in red

Once again, we plan to have problets automatically generate this specification while executing the program, by keeping track of the lines of code in which a particular variable appears.

Clustering code: When analyzing the behavior of a program, clustering together logically related lines of code greatly helps comprehension, e.g., drawing boxes around each loop in a program with nested loops (e.g., see [6]). Overlay visualization can be used for this purpose. Whereas code and data flow visualization map time onto space, i.e., map behavior of the program at discrete events of time on to the text of the code, clustering maps space onto space, i.e., maps discrete lines of code into related clusters.

4 Use of Overlay Visualization in Problets

We will now describe the use of overlay visualization for presenting problems, automatically solving them, entering answers and grading in problets.

Consider the following C++ code presented by a problet on variables:

```
void main()
{
   int index = 855;
```

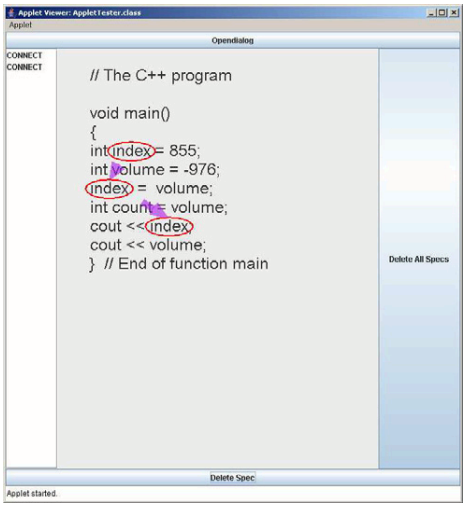

Fig. 4. An Example of Data Flow Visualization being generated by the student.

```
    int volume = -976;
    index = volume;
    int count = volume;
    cout << index;
    cout << volume;
}
```

The problet knows that the variable `index` is declared, assigned and referenced on lines 5, 7, and 9 in the program. Based on this information, it automatically generates the following visual specifications:

(i) CONNECT line 5 to line 7

(ii) CONNECT line 7 to line 9

The visualization corresponding to these specifications, which is automatically generated by the problet, is shown in Figure 5. The problet prompts the student to visualize the data flow of the variable `index` using mouse gestures. The student connects line 5 and 7 with a click-and-drag gesture, and lines 5 and 10 with another click-and-drag gesture as shown in Figure 4. The problet translates these gestures to the following visual specifications:

(i) CONNECT line 5 to line 7

(ii) CONNECT line 7 to line 10

Finally, the problet compares the correct visual specifications with the specifications generated by the student through mouse gestures, and provides the following feedback:
You have correctly connected line 5 and 7
You have incorrectly connected line 7 and 10

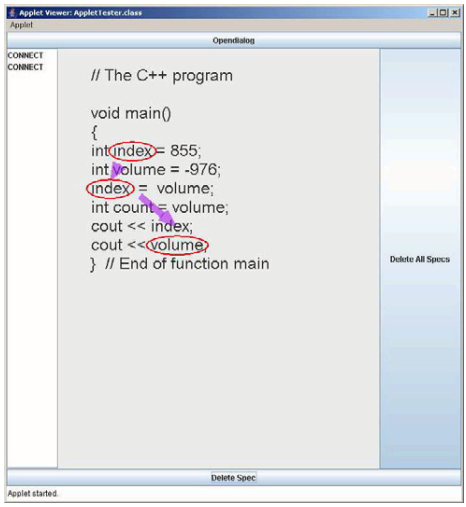

Fig. 5. An Example of Data Flow Visualization automatically generated by problets.

5 Discussion

There are several advantages to using overlay visualization:

- **Less Cognitive load:** There is less cognitive load if both the code and the visualization are displayed on the same panel. The student doesnt have to alternate between two separate panels, and mentally put together the information from the two panels. His/her attention will be focused on both the code and the visualization at the same time.
- **Execution History:** As compared to code visualization, where only the currently executing line of code is highlighted (e.g., [8]), overlay visualization automatically supports the display of execution history. Including execution history is one of the ways to improve the effectiveness of visualization [12]. In other words, overlay visualization supports path visualization (execution history), just as easily as state visualization (a current snapshot of the program being executed).
- **Active learning:** One of the recommendations for improving the effectiveness of visualization is to let the learner construct his/her own visualization [12], [4]. This will promote visual as well as active learning. With overlay visualization, students can construct their own visualization using mouse gestures. For example students can specify Connect and Point by dragging the mouse, highlight by double-clicking the mouse, etc. The user interface can translate mouse gestures into visualization specifications, which can then be compared with the correct specifications to provide feedback to the learner.

Overlay visualization is meant to be used as a supplement to, and not a substitute for the traditional types of visualization used for program analysis, such as data space visualization (snapshot of all the variables and their values), data flow visualization (highlighting the flow of data from one variable/object to another), code visualization (highlighting the line of code that is currently being executed),

control space visualization (flowchart of the program, call graph, UML diagram, etc.), and control flow visualization (highlighting the path of execution of a program) provided in visual debuggers (e.g., Retrovue http://www.retroview.com/, whyline [5]), program visualizers (e.g., Jeliot 3 [11]) and concept visualizers (e.g., [3] [9]).

Overlay visualization is proposed as a tool, not as a specific type of program visualization - as a matter of fact, it can be used for many of the traditional types of program visualization mentioned above. We believe that it is especially effective for data flow and control flow visualization since it superimposes the visualization on the program code, thereby reducing the student's cognitive load. Overlay visualization is not even restricted to program visualization - it can be used to connect the sequence of arguments/stream of thought in text, highlight clues in a puzzle, highlight the structure of a web page, etc. Meta-data can be used to automatically generate such overlay visualization.

Representation of the program text is the most basic form of program comprehension [14]. Since overlay visualization is superimposed on program text, it is especially suitable for novice programmers. Traditionally, program visualization systems visualize the program for the student. With overlay visualization, it is also easy to have the student specify the visualization using two of the most basic facilities: the program text, which is the most basic form of representation of the program (as compared to data space, flowchart, class graph, UML diagram, etc. which require a deeper understanding of the program), and mouse gestures, which are the most primitive form of user interaction. So, overlay visualization is especially amenable to active learning, and for use by novice programmers.

We plan to evaluate the effectiveness of overlay visualization in problets in fall 2006.

Acknowledgement

Partial support for this work was provided by the National Science Foundations Educational Innovation Program under grant CNS-0426021, and by Student Activities Revenue Management of Ramapo College of New Jersey.

References

[1] Baggett, P., *Role of temporal overlap of visual and auditory material in forming dual media associations*, Journal of Educational Psychology **76** (1984), pp. 408–417.

[2] Ben-Ari, M., N. Myller, E. Sutinen and J. Tarhio, *Perspectives on program animation with jeliot: In diehl, s. (ed.)*, Software Visualization. LNCS **2269** (Springer-Verlag, 2002), pp. 31–45.

[3] Bowdidge, R. W. and W. G. Griswold, *Supporting the restructuring of data abstractions through manipulation of a program visualization*, ACM Trans. Software. Engineering. Methodology **7** (1998), pp. 109–157.

[4] Hundhausen, C. D., S. A. Douglas and J. T. Stasko, *A meta-study of algorithm visualization effectiveness*, Journal of Visual Languages and Computing **13** (2002), pp. 259–290.

[5] Ko, A. J. and B. A. Myers, *Designing the whyline: a debugging interface for asking questions about program behavior*, In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vienna, Austria, CHI '04. ACM Press, New York, NY (2004), pp. 151–158.

[6] Kumar, A., *Generation of problems, answers, grade and feedback - case study of a fully automated tutor*, Journal of Educational Resources in Computing (JERIC) (2006a).

[7] Kumar, A. and S. Kasabov, *Observer architecture of program visualization*, Proceedings of the Fourth Program Visualization Workshop, Florence, Italy (2006).

[8] Loboda, T., A. Frengov, A. Kumar and P. Brusilovsky, *Distributed framework for adaptive explanatory visualization*, Proceedings of the Fourth Program Visualization Workshop, Florence, Italy (2006).

[9] Malloy, B. A. and J. F. Power, *Exploiting uml dynamic object modeling for the visualization of c programs*, In Proceedings of the 2005 ACM Symposium on Software Visualization (2005), pp. 105–114.

[10] Mayer, E. and R. Anderson, *Animations need narrations: An experimental test of a dual-coding hypothesis*, Journal of Educational Psychology **83** (1991), pp. 484–490.

[11] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with jeliot 3*, In Proceedings of the Working Conference on Advanced Visual interfaces, Gallipoli, Italy, ACM Press, New York, NY (2004), pp. 373–376.

[12] Naps, T., R. Fleischer, M. McNally, G. Rossling, C. Hundhausen, S. Rodger, V. Almstrum, A. Korhonen, J. Velazquez-Iturbide, W. Dann and L. Malmi, *Exploring the role of visualization and engagement in computer science education.*, SIGCSE Bulletin **35** (2003), pp. 131–152.

[13] Paivio, A., *Mental representations: A dual coding approach*, New York: Oxford University Press (1990).

[14] Pennington, N., *Stimulus structures and mental representations in expert comprehension of computer programs*, Cognitive Psychology **19** (1987).